

Discovering Vulnerabilities In The Wild: An Empirical Study

by

Ming Fang

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 13, 2014

Keywords: Secure Software Engineering, Vulnerability, Empirical Study

Copyright 2014 by Ming Fang

Approved by

Munawar Hafiz, Assistant Professor of Computer Science and Software Engineering
Hari Narayanan, Professor of Computer Science and Software Engineering
Jeffrey Overbey, Assistant Professor of Computer Science and Software Engineering

Abstract

There is little or no information available on what actually happens when a software vulnerability is detected. We performed an empirical study on reporters of the three most prominent security vulnerabilities: buffer overflow, SQL injection, and cross site scripting vulnerabilities. The goal was to understand the methods and tools used during the discovery and whether the community of developers exploring one security vulnerability differs—in their approach—from another community of developers exploring a different vulnerability. The reporters were featured in the SecurityFocus repository for 12 month periods for each vulnerability. We collected 127 responses. We found that the communities differ based on the security vulnerability they target; but within a specific community, reporters follow similar approaches. We also found a serious problem in the vulnerability reporting process that is common for all communities. Most reporters, especially the experienced ones, favor full-disclosure and do not collaborate with the vendors of vulnerable software. They think that the public disclosure, sometimes supported by a detailed exploit, will put pressure on vendors to fix the vulnerabilities. But, in practice, the vulnerabilities not reported to vendors are less likely to be fixed. Ours is the first study on vulnerability repositories that targets the reporters of the most common security vulnerabilities, thus concentrating on the people involved in the process; previous works have overlooked this rich information source. The results are valuable for beginners exploring how to detect and report security vulnerabilities and for tool vendors and researchers exploring how to automate and fix the process.

Acknowledgments

Put text of the acknowledgments here.

Table of Contents

Abstract	ii
Acknowledgments	iii
List of Figures	viii
List of Tables	x
1 Introduction	1
1.1 Thesis Statement	3
1.2 Contributions	4
1.3 Organization of Document	4
2 Background	6
2.1 Vulnerability Repositories	6
2.2 Studying Vulnerability Repositories	7
2.3 Understanding Security Engineering Tools and Practices	9
3 Research Methodology	12
3.1 Scope of the Study	12
3.2 Participants	13
3.3 Questionnaire	14
3.4 Process	15
3.5 Coding	16
4 Results	18
5 Distribution of Participants	19
5.1 Many years of vulnerability detection experience	19
5.2 Single to a thousand vulnerabilities	20
5.3 A blend of professionals and hobbyists	21

5.4	Reporters explore different variants of vulnerabilities	22
6	Study on Reporters of Buffer Overflow Vulnerabilities	23
6.1	RQ1: General Approach	23
6.1.1	Reporters start blindly	23
6.1.2	Reporters actively probe popular targets	24
6.1.3	Windows binaries and Unix source code	25
6.1.4	Fuzzing is the chosen method	26
6.1.5	Applying static analysis is uncommon	26
6.1.6	Code review and other manual approaches are useful in detecting vul- nerabilities in source code	27
6.2	RQ2: Use of Tools	27
6.2.1	Off-the-shelf fuzzing tools are typically not used	27
6.2.2	Immunity Debugger, IDA Pro, and WinDbg in Windows platform, GDB in Unix platform	29
6.2.3	Static analysis tools are almost never used	30
6.3	RQ3: Vulnerability Reporting	30
6.3.1	A lot of reporters prefer full disclosure	30
6.3.2	Experts choose highly visible reporting options	31
6.3.3	Fully disclosed vulnerabilities may remain unfixed	32
6.3.4	Exploit Database (EDB) is the most popular	32
6.4	RQ4: Exploit Generation	33
6.4.1	Reporters typically write exploits	33
6.4.2	Perception of exploit generation effort varies	33
6.4.3	Exploits are written to make public	34
7	Study on Reporters of SQL Injection Vulnerabilities	35
7.1	RQ1: General Approach	35
7.1.1	Reporters explore for bugs in software that they use	35

7.1.2	Reporters target applications for specific reasons	36
7.1.3	Most reporters explore for SQL injection vulnerabilities manually . . .	36
7.2	RQ2: Use of Tools	38
7.2.1	Reporters mention using penetration testing tools	38
7.2.2	Static analysis tools are almost never used	38
7.3	RQ3: Vulnerability Reporting	39
7.3.1	A lot of reporters prefer full disclosure	39
7.3.2	Fully disclosed vulnerabilities may remain unfixed	39
7.3.3	Exploit Database (EDB) is the most popular	40
7.4	RQ4: Exploit Generation	40
7.4.1	Reporters typically write exploits	40
7.4.2	Exploits are written to make public	41
8	Study on Reporters of Cross Site Scripting Vulnerabilities	42
8.1	RQ1: General Approach	42
8.1.1	Reporters explore for bugs in software that they use	42
8.1.2	Reporters stated various reasons about why they target an application	43
8.1.3	Most reporters explore for cross site scripting vulnerabilities manually	43
8.2	RQ2: Use of Tools	44
8.2.1	Reporters mention using penetration testing tools	44
8.2.2	Static analysis tools are almost never used	45
8.3	RQ3: Vulnerability Reporting	45
8.3.1	Reporters prefer to report to vendors	45
8.3.2	No specific repository is popular	45
8.4	RQ4: Exploit Generation	46
8.4.1	Reporters typically write exploits	46
9	Communities Formed Around Vulnerabilities	47
9.1	Different reasons to select a target application	47

9.2	Buffer overflow reporters use tools, other reporters detect manually	48
9.3	Cross site scripting reporters reported to vendors more	50
10	Discussion and Recommendations	51
11	Threats to Validity	55
12	Future Work and Conclusion	57
	Bibliography	58

List of Figures

3.1	Steps of Data Collection Process	15
3.2	Steps of Data Analysis Process	16
5.1	Experience of Reporters (Years Working On Vulnerabilities)	20
5.2	Experience of Reporters (Reported Vulnerabilities)	20
6.1	Context of Discovery	24
6.2	Platforms and Code	24
6.3	Popularity of Fuzzing	25
6.4	Custom Fuzzing Tools	25
6.5	Tools In Use	28
6.6	Reporting Practices	28
6.7	Exploit Generation	33
6.8	Which Problems Are Fixed?	33
7.1	Context of Discovery	37
7.2	Reporting Practices	37
7.3	Exploit Generation	39

7.4	Which Problems Are Fixed?	39
8.1	Context of Discovery	43
8.2	Reporting Practices	43
8.3	Exploit Generation	46
8.4	Which Problems Are Fixed?	46
9.1	Set of Reporters Targeting Different Vulnerabilities	48

List of Tables

3.1	Study Participants	13
3.2	Study Questions For Participants	14
9.1	Reporters With Access To Source Code	49
9.2	Vulnerability Reporting Practices	50

Chapter 1

Introduction

A software vulnerability is a security weakness in the software program that provides opportunities for malicious attacks to reduce a system's information assurance [39], and software vulnerabilities have broad impact and may bring the significant damage to vendors and users [34]. Therefore, it is important for security professionals to do research on detecting, disclosing and fixing software vulnerabilities.

There are three prominent classes of vulnerability—buffer overflow vulnerability, SQL injection vulnerability, and cross site scripting (XSS) vulnerability—that affect most software.

A buffer overflow vulnerability occurs when data is written into a program buffer and exceeds the assigned size of the buffer. As a result, a buffer overflow vulnerability can be triggered by the designed input to cause program issues, such as system security, wrong data, memory access errors and administrative issues.

A SQL injection vulnerability includes insertion or injection of a designed SQL query to the input data from the client to the application. Attackers can attain sensitive data from the database, modify database data, and execute administration operations on the database by SQL injection vulnerabilities.

A cross site scripting (XSS) vulnerability is a type of injection that injects malicious scripts into trusted web sites. XSS vulnerabilities occur when attackers inject malicious code by web applications in the form of a browser side script to end-users. At last, scripts will be downloaded to users' browsers and executed after users visit these web applications.

These vulnerabilities have drawn lots of researchers to design algorithms and develop tools to detect and report them. It often seems that a practitioner faces too many choices,

but he/she does not have sufficient guidelines. For example, Beth, a beginner, wants to find the information about how to detect buffer overflow vulnerabilities, she can pursue many sources of information. She can search the Internet, ask her peers, read books and research papers on detecting overflows, or even train to become an ethical hacker. She will be easily overwhelmed by the number of “best practices” suggested and the number of suggesters claiming that their methods work best without backing the claims with empirical data. A researcher who wants to improve the security engineering process to fix the reported vulnerabilities more efficiently will also need empirical data to understand which problems to target. A similar information gap confuses tool builders who want to build popular, widely-adopted tools.

Even when an expert suggests a method, it is based on what has worked for him/her; other experts may differ. Perhaps, there is a method that works really well for detecting a kind of vulnerability; perhaps, this method is used by many security experts; but there have been no studies to understand and document this trend.

Scholte [29] argued when a security vulnerability exists in the software, there would be a window gap of disclosure that can exist until the vulnerability is fixed and the patch is installed by users. The number of engineers who can exploit the vulnerability and the time that it will take to be patched decides the shape of this window. Researchers who have studied security engineering process focused on finding statistics of vulnerability trends, lifecycle of vulnerabilities, time to fix a vulnerability, etc [16–18,28,29]. These studies, however, have not focused on the approaches of the people who report buffer overflow vulnerabilities, i.e., *the tools they use and methodologies they follow*.

Also, previous studies explore the tip of an iceberg; they miss a lot of activities going on behind vulnerability detection and reporting. *The people involved in these activities carry first-hand information about corresponding security approaches*. The knowledge that they possess is not stored in vulnerability repositories or any other repositories. No attempts have been made to tap into this rich information source. If the connections can be built

with reporters and they can supply more details that can't be found from books, papers or the Internet, such as vulnerability detecting methods, tools, vulnerability disclosure strategy and exploit generation, this would give us potentially better guidelines.

This thesis describes a study performed on *reporters* of buffer overflow vulnerabilities, SQL injection vulnerabilities, and cross site scripting vulnerabilities featured at SecurityFocus repository [31]. Our analysis is based on the large number of responses of 58 reporters of buffer overflow vulnerabilities (out of 229 with contact information, 25.33% response rate), 27 reporters of SQL injection vulnerabilities (out of 131 with contact information, 20.61% response rate, and 42 reporters of cross site scripting vulnerabilities (out of 227 with contact information, 18.50% response rate). The group of responders included reporters who have discovered a vulnerability for the first time as well as reporters who have reported hundreds of vulnerabilities, reporters who have a few months experience as well as reporters with over ten years of experience, reporters working on source code as well as reporters working on binaries.

We asked them about the *methods* they followed to detect and report vulnerabilities and the *tools* they used. The questions were open-ended, so reporters can provide more specific details. We analyzed the responses by applying a structural coding approach [27] using Text Analysis Markup System Analyzer (TAMS) [37], an open source tool for coding. TAMS can help us cycle the keywords from large paragraph of detail and convert similar answers to the same key word by special taggings to help us find common trends from these data.

1.1 Thesis Statement

Collecting information from reporters of security vulnerabilities reveals value insight about vulnerability detection practices that are not to be found in vulnerability repositories. More details can be found about the methods people follow, the tools they use, the way they interact with vendors, and report vulnerabilities, and the effort they put in creating exploits. However each specific vulnerabilities nurtures its own community.

1.2 Contributions

Our goal was to research what people do to detect these three kinds of vulnerabilities, find the common trends and different features among various kinds of vulnerabilities, and capture their essence by analyzing the data separately and across.

Our research makes the following contributions.

1. To the best of our knowledge, it describes the first study to collect information about practices from the people involved in discovering three different significant types of vulnerabilities: the most reliable, yet previously ignored, source of information. The data is collected from 58 reporters of buffer overflow vulnerabilities, 42 reporters of cross site scripting vulnerabilities, and 27 reporters of SQL injection vulnerabilities that were featured in the SecurityFocus repository during two same length periods.

2. It describes a research methodology that can be reused by researchers who want to understand the discovery process of other security vulnerabilities.

3. It presents information about what actually happens during the detection, analysis, and reporting of buffer overflow vulnerabilities and web-related vulnerabilities. It suggests which approaches are used, which work, which do not, and what needs to changes.

4. It compared the data of buffer overflow vulnerabilities and web-related vulnerabilities, and also analyzed the data across to reveal the common trends and the different features among these three types of vulnerabilities.

5. It showed some ideas for the future work of software vulnerability research.

1.3 Organization of Document

Chapter 2 describes the background of our research. It introduces the study of vulnerability repositories, and security engineering tools and practices. Chapter 3 presents the research methodology including scope of the study, participants, questionnaire, process and coding. Chapter 4 formulates four categories of research questions to structure the study

and the questions we asked the reporters. Chapter 5 shows the distribution of participants of these three kinds of vulnerabilities. Chapter 6, 7, and 8 analyze the data of buffer overflow vulnerabilities, SQL injection vulnerabilities and cross site scripting vulnerabilities separately. Chapter 9 gets common trends and different features among these three kinds of vulnerabilities. Chapter 10 gives discussion and recommendations. Chapter 11 shows threats to validity. The conclusions and future work are described in Chapter 12 .

Chapter 2

Background

This section describes previous work on studying vulnerability repositories and studies on human subjects that focus on understanding security engineering tools and practices.

2.1 Vulnerability Repositories

There are hundreds of vulnerability repositories that track various software vulnerabilities. All these vulnerability repositories can be classified by public or private, free or not free, and third party or vendor. However, lots of repositories miss vulnerabilities' details and are out of date, which reduces the probability to be accessed by researchers. There are some popular repositories and each of them has its own specific features [22].

1. National Vulnerability Database (NVD): A U.S. government repository that uses the Security Content Automation Protocol (SCAP) to represent standards-based vulnerability management data.

2. SecurityFocus: An online computer security bulletin and purveyor of information security services and home to the well-known Bugtraq mailing list. One of the most comprehensive (more than 30000 entries).

3. Exploit Database (EDB): A repository of exploits and vulnerable software, and its main goal is to collect exploits from mailing lists to focus on them in a easy to navigate repository.

4. Open Source Vulnerability Database (OSVDB): An independent open source vulnerability repository to provide comprehensive information regarding vulnerabilities like solutions, links to patches, affected products etc.

5. Zero Day Initiative (ZDI): An platform to pay researchers for responsibly disclosing vulnerabilities.

6. Microsoft Security Bulletin: The security bulletin for products from Microsoft, e.g., Windows, Internet Explorer, etc.

7. Mozilla Foundation Security Advisories(MFSA): The platform for reporting Mozilla products.

8. Bugzilla: A web-based application that keeps track of programming bugs.

The most complete vulnerability repository should combine the data from all vulnerability repositories. However, it is not easy to create a more complete data set by collecting the data from several repositories at the same time and there is not always a common factor among vulnerability descriptions from each repository that we could use easily. It is also difficult to match vulnerabilities from different repositories, because CVE references have not been attained by all vulnerabilities. Therefore, we have to select the vulnerability repository that shows most convenient and helpful for our research before collecting and analyzing data [22].

2.2 Studying Vulnerability Repositories

Studies on vulnerability repositories focus on harvesting statistical trends [3,18,28,29,32] or creating vulnerability models and using them for prediction [1,2,9].

Schneier [28] first focused on the entire window of disclosure to a vulnerability and proposed a life cycle model for vulnerabilities; later Arbaugh et al. [3] proposed a modified model. Most of the studies of vulnerability repositories consider this life cycle model (or an adapted version) and reach statistical conclusion on various aspects, e.g., time to create a patch. Arora and colleagues [4] studied 308 vulnerabilities to find out how efficiently vendors respond to vulnerability reports. Frei and colleagues [17] studied Microsoft and Apple patches from CERT, Secunia, SecurityFocus, and some other repositories to measure the time to produce a typical patch.

Frei et al. [16] proposed a modified vulnerability life cycle model later. They introduced definitions of significant time periods, such as time of exploit generation, time of patch development, time of patch installation. They also provided the first examination of impact and risks combined with this gap, and a metric for the success of the responsible “disclosure” process.

Other researchers concentrated on the trends in vulnerabilities. Gopalakrishna and Spafford [18] studied CVE reports of five well-known software to find a connection of vulnerability types reported on the same software. Their results suggest that the discovery of a vulnerability in a software may influence the discovery of the same type of vulnerabilities in that software. Scholte and colleagues [29] studied the National Vulnerability Database (NVD) to analyze how cross site scripting and SQL injection vulnerabilities evolved over time in web applications. Cova and colleagues [10] searched various malware hosting sites as opposed to vulnerability repositories. They focused on a separate type of trend: how infrastructures, e.g., web server, DNS server, etc., spread rogue antivirus software. Wu and colleagues [39] studied how semantic templates can be overlaid on vulnerability repositories to harvest more systematic information; this can be useful for further trend analysis.

Shahzad and colleagues [32] studied 46,310 vulnerability reports from three repositories including NVD and Open Source Vulnerability Database, OSVDB, from 1998 to 2011 and analyzed trends such as the prominence of buffer overflow and DoS vulnerabilities, the increase of remotely exploitable vulnerabilities, etc. They found vendors can develop patches for vulnerabilities faster and the access complexity of vulnerabilities has been increasing, but attackers can spend less time than vendors to exploit a vulnerability. They also believed vulnerability patches for open source are much slower than those for closed source, but exploits of open source program can be faster.

Many researchers have focused on creating vulnerability models and using them for prediction. Browne and colleagues [9] first studied CERT database for vulnerabilities reported on some well-known software in search of a trend of exploits.

Anbalagan and Vouk [2] created a model to describe connections between security vulnerabilities and security exploits. They quantified vulnerability disclosure rate, exploit generation and patching rates by analyzing the available data. They also discussed an approach to understand security problem exploit connection and the impact of those exploits to correct vulnerabilities issues. At the same time, they treated the system as a whole and a single system by checking time periods.

Alhazmi and colleagues [1] presented a vulnerability discovery model to estimate the number of vulnerabilities for different operating systems. They collected data from several operating systems, such as Windows 98, Windows 2000 and Red Hat Linux 7.1 and checked the impact of taking a specific constraint, and found the estimation error rate is very low when a constraint based on past observations is added. Their results suggested combining static and dynamic analysis may be possible to improve the estimation capability.

In contrast, Zhang et al. [41] implemented data mining and machine learning technologies to find a trend in vulnerabilities reported in the NVD database unsuccessfully. They found that data from NVD generally usually have poor estimation capability for vulnerabilities, and it only works well for a few vendors and software programs.

Our study focuses on the reporters who possess the most important information; no previous works have explored this source. Researchers have done lots of research of vulnerability collecting, detecting, exploiting, and reporting, and their results provide active help for our research on the empirical study of discovering these three types of vulnerabilities.

2.3 Understanding Security Engineering Tools and Practices

This paper focuses on the human factor in vulnerability detection, which is missing in most of the research. Only a few researchers have studied human factors of security engineering. Schryen [30] started a survey of a collection of client and server side software to analyze vendors' patching behavior. His research focused on the first comprehensive empirical study on the security of open source and closed source. Their results revealed the

open source and closed source programs are not significantly different in respect of vendors' patching behavior. He believed it is important for vendors to provide strong economic incentives to supply patches and keep the published vulnerabilities being fixed fast.

Okhravi and Nicol [24] studied browser vulnerabilities to understand how much pre-deployment testing is optimal. They tested the overall security of the system under different operation circumstances and estimated the adjustment between test time and the vulnerability disclosure window by using simulation models, patch development and deployment processes.

There have been a few studies comparing tools and approaches to detect vulnerabilities. Layman et al [21] conducted a user study to design better software vulnerability detection tools. Their study revealed several important factors that can affect developers' decision to interrupt program development debug the vulnerability issues with the detecting tool. Wilander and Kamkar [38] studied four compiler based approaches to detect and prevent buffer overflow vulnerabilities, but these tools only provide a partial solution. They proposed several run-time techniques to detect and prevent most common vulnerabilities, but none of them can detect the various known vulnerabilities today.

Studies on automated penetration testing tools on web vulnerabilities have also reported that existing tools missed many vulnerabilities [5, 12, 33]. Among these works, Austin and colleagues [5] reported that a combination of different approaches (manual analysis, static analysis, and automated penetration testing) is necessary to discover vulnerabilities.

These studies imply that some approaches are better than others in detecting vulnerabilities; they do not focus on whether the tools and approaches are used by people in practice.

Our study found that reporters rarely use static analysis tools when they search for vulnerabilities. This justifies more work on understanding the challenges of using static analysis in practice. Works by Johnson and colleagues [19], Baca and colleagues [6], and Rutar and colleagues [26], explored the usability challenges of static analysis tools.

There are two main reasons for developers to select the static analysis tool to detect vulnerabilities or bugs. First, manual detecting vulnerabilities will cost too much time and effort, while the static analysis tool can detect vulnerabilities much faster and easier. Second, the static tool is always already available on the shelf and ready to be used. However, several factors including tool output, collaboration, and coverage may stop developers using static analysis tools, while some detectors would like to select these tools to detect vulnerability.

Beca et al. [6] found the automatic static analysis tool was capable to detect memory related vulnerabilities, but it did not work well on few other types of vulnerabilities. The deployment of tools played an important role in their success as early vulnerability detecting tools, however, static analysis tools should be combined with vulnerability reporting systems and developers also should share the responsibility for classifying and reporting warnings. Rutar and colleagues [26] started a comparison of vulnerability detecting tools for java program. Even though there are some overlaps among vulnerabilities detected by those tools, detecting tools' warnings are almost distinguished and the main difference among these tools is the number of output. Developers need to add an annotation into the code to prevent warnings with false positives, although it might bring some potential issues.

These works just focused on the needs of secure software developers, not on the specific needs of reporters of security vulnerabilities.

Chapter 3

Research Methodology

This section narrates our research methodology in detail. We describe the scope of the study, how the participants were selected, which questions were asked, how we collected data, and how we analyzed the responses. Figures 3.1 and 3.2 show the data collection and analysis steps respectively.

3.1 Scope of the Study

We conducted our study on all reporters of the target vulnerabilities that were featured in the SecurityFocus repository during twelve-month periods. For buffer overflow, we started with vulnerabilities reported between December 1, 2011 to May 31, 2012, and December 1, 2012 to May 31, 2013. For SQL injection and cross site scripting vulnerabilities, we started with vulnerabilities reported between November 1, 2012 to October 31, 2013. Not all of these vulnerabilities were new: SecurityFocus lists vulnerabilities reported earlier if some new information is added to the listing.

Choosing the right repository is important for the success of the study. Massacci and Nguyen [22] studied the quality of a vulnerability database by measuring how often it is used by other security researchers.

We chose SecurityFocus [31] repository because it is well-known (Section 6.3.4), it is highly ranked in Massacci and Nguyen’s [22] study, and we have used this repository in previous research. Massacci and Nguyen also report that it is one of the most widely used repositories in empirical studies only behind the vulnerability list kept by National Vulnerability Database (NVD). We chose SecurityFocus over NVD because it contains more vulnerabilities than NVD. For example, SecurityFocus listed 516 vulnerabilities with at least

	# of Vulnerabilities (Col. 1)	No Reporters Attributed (Col. 2)	Vulnerability w/ Reporters (Col. 3 = 1-2)	# of Reporters (Col. 4)	No Email Info. Found (Col. 5)	Incorrect Email Info (Col. 6)	# of Reporters Reached (Col. 7 = 4-5-6)	# of Reporters Responded (Col. 8)	% Responded (Col. 9 = 8/7)
Buffer Overflow	623	107	516	332	78	25	229	58	25.33%
SQL Injection	545	91	454	210	58	21	131	27	20.61%
Cross Site Scripting	947	279	668	457	204	26	227	42	18.50%

Table 3.1: Study Participants

one reporter during the study period when we focused on buffer overflow reporters, some new and some updates to old vulnerabilities. Only 390 (75.58%) had a Common Vulnerability Enumeration (CVE) identifier; these were in NVD. Choosing SecurityFocus allowed us to contact with more people and potentially collect more responses.

3.2 Participants

Figure 3.1 shows the steps of the data collection process and Table 3.1 shows how the study participants were chosen. We started with all the vulnerabilities of a specific kind featured during the study period (e.g., 623 buffer overflows). Some vulnerabilities do not have any reporter associated, while some others are reported by vendors of the corresponding software. These are aggregated in column 2. The remaining vulnerabilities (column 3) attribute at least one reporter; some have two or three reporters.

Table 3.1 lists the number of reporters in column 4. Some of them reported multiple vulnerabilities during the study period. The vulnerability reports attribute reporters, but do not store their contact information. We searched for the contact information from various sources, e.g., multiple vulnerability repositories, security advisories, emails in vendor webpage where bugs were reported, etc. In many cases, we searched for the reporters on the Internet using their names.

We could not identify the email address of some reporters for various reasons (column 5). For some other reporters, the email address that we found turned out to be incorrect (column 6). The remaining reporters received our questionnaire. The last two columns show

the number of reporters responding to our study and the response rate. Reporters of buffer overflow vulnerabilities had a higher response rate than others.

3.3 Questionnaire

Questionnaire
<p><i>[■ is specific question for SQL Injection Vulnerability, □ is specific question for XSS Vulnerability]</i></p>
<ul style="list-style-type: none"> ■ 1. Which SQL statement did you find vulnerable? SELECT, INSERT, UPDATE, DELETE? Was it a second order SQL injection? □ 1. What is the type of the vulnerability that you reported (e.g., XSS, CSRF, HTML injection, clickjacking, etc.)? ■□ 2. Is this the first security vulnerability that you have reported? If not, how many previous vulnerabilities have you reported? 3. Do you focus on web-related vulnerabilities only? What other kinds of vulnerabilities (e.g., buffer overflow) did you report? 4. How long have you been interested in security vulnerabilities? Do you do this as a hobby, or is it your professional responsibility? 5. Please comment on your familiarity with the application. Were you affiliated with the development process of the application, or were you an end user? 6. Why did you select this application? Was it a random selection? 7. Was the detection of this vulnerability a coincidence or were you specifically searching for vulnerabilities? 8. Did you have access to the source code when you discovered the vulnerability? Or were you working on binaries only? 9. Can you please describe the steps you followed while detecting the vulnerability? Do you follow these same steps for other targeted vulnerabilities you may have detected? 10. Were any tools used to discover this vulnerability? Please name them and describe how these tools were used to detect this vulnerability. 11. If you used tools, why did you choose these tools? 12. How did you first find out about these tools? Magazines? Web Search? Suggestion from a peer? 13. Have you used other tools for detecting other instances of buffer target (or related) vulnerabilities? Please name them. Why were they not used here? 14. Did you prepare an exploit demonstrating the vulnerability? If yes, can you provide a general description of the process? What tool(s) did you use? How much time did it take? 15. Which forum did you use to report the vulnerability and why did you select this particular forum? 16. Did you search for additional vulnerabilities in the same application once this vulnerability was detected? Did you find/report any more?

Table 3.2: Study Questions For Participants



Figure 3.1: Steps of Data Collection Process

Table 3.2 lists the questions asked. The buffer overflow study had 14 questions. Questionnaire for reporters of SQL injection and cross site scripting vulnerabilities repeated most of these questions; only two new questions were added that were specific to these vulnerabilities (questions 1 and 3 in Table 3.2).

First few questions ask about a participant’s background. Questions 5–9 and 16 focus on the methodology followed by a reporter, while question 10–13 focus on the tools used. Questions 14 and 15 explores reporting and exploit generation tasks.

3.4 Process

The buffer overflow study was done in two rounds, each covering vulnerabilities reported for a six-month period; the study on the other two vulnerabilities covered a twelve-month period. We sent emails to the reporters whose address could be found. The emails were sent as we processed the information about vulnerabilities and identified reporters. Typically 10-15 emails were sent per week. We sent a reminder typically once a month and stopped after three reminders. A few reporters who reported more than one kind of vulnerability received one email for each kind of vulnerability.

The email we sent contained the questions and a reference to a specific vulnerability. We referred to the vulnerability with the BugTraq ID (BID) used in SecurityFocus and the

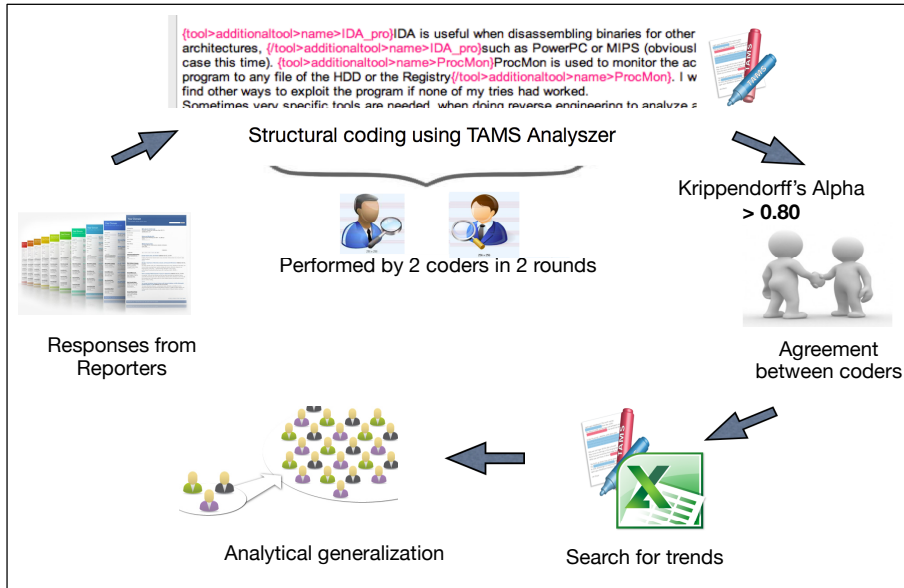


Figure 3.2: Steps of Data Analysis Process

CVE identifiers used by NVD, when applicable. For reporters of multiple vulnerabilities, we listed all vulnerabilities reported, but asked him/her to focus on the last vulnerability reported.

Pilot Study. Before the buffer overflow study, we launched a small pilot study to train ourselves with the process and finalize the questions to ask. We selected 10 reporters of buffer overflow vulnerability from the SecurityFocus repository during the first week of November 2011. Each reporter received the initial version of the questionnaire containing 9 questions.

We received 4 responses. The responses helped us adapt our study. For example, one responder suggested that we include CVE identifiers of a vulnerability as well as the BugTraq ID (BID) used by SecurityFocus. Also, we restructured the questionnaire based on the suggestions [14].

3.5 Coding

Figure 3.2 shows the steps we followed to analyze the data. We applied structural coding [27] to annotate the responses. Having specific and well-defined categories to code

also reduced interpreter's bias. The codes we selected had clear partitions. For example, when a responder says that he/she has six years of experience, and we had codes that defines experience as 'less than one year', 'one to less than five years', 'five to less than ten years', and 'more than ten years', there is only one code that can be applicable.

We used TAMS Analyzer [37], an open source tool, for coding as well as analyzing the codes. Both authors applied codes to all responses in TAMS Analyzer. To ensure reliability, their codes were compared. Because of structural coding, the codes mostly matched (e.g., initial Krippendorff's alpha [20] value for the buffer overflow study was 0.8339). The coders analyzed the codes that differed. Some mismatches were from one coder missing a code. Most mismatches were because of the limits of TAMS Analyzer: if two codes were applied to the same paragraph in a different order, TAMS Analyzer identified them as different. We resolved all the issues to reach consensus.

Chapter 4

Results

Most of the results presented in this paper come from applying analytic generalization [40]. While we present some statistical information about the tools and methods used by reporters (statistical generalization [40]), these only summarize the findings. We present the results as aggregates of the two rounds of study.

We formulated four categories of research questions to structure the study and the questions we asked the reporters.

RQ1: General Approach: Are vulnerabilities discovered by chance or is there a general approach that reporters follow? How does a reporter select a target application? Are there different approaches when people work with source code or binary?

RQ2: Use of Tools: Are there tools that are commonly used? Why and how are they used? How is the information about these tools disseminated among reporters?

RQ3: Vulnerability Reporting: How and where are the vulnerabilities reported? Do reporters work with the vendors to fix vulnerabilities? What are the consequences of the choices made by reporters?

RQ4: Exploit Generation: Do reporters write exploit code after they detect vulnerabilities? What types of tools are used to create exploit code?

We first describe the distribution of participants (Section 5). Then we describe the findings for the three studies, each time focusing on the four categories of research questions (Sections 6—8). This will be followed by a discussion of the separate communities that are formed by the reporters of these vulnerabilities.

Chapter 5

Distribution of Participants

This section describes that the participants represent reporters with different experience levels (years of experience), levels of success (vulnerabilities reported), types of involvement (professional or hobbyist), and working contexts (type of target software, operating system, etc.). This validates the analytic generalizability of the responses. Also, we can analyze the data from different perspectives, e.g., how do experts do X (e.g., buffer overflow reporting practices, Section 6.3), or what do developers who target Unix-based systems do for X (e.g., tools used by buffer overflow reporters, Section 6.2), etc.

5.1 Many years of vulnerability detection experience

Figure 5.1 shows the experience of reporters in terms of the number of years they have been involved in this effort. For buffer overflow vulnerability, 43 out of 58 reporters responded to the question about their experience with a precise number of years. The group includes beginners as well as experts. Only 2 reporters had less than one year of experience, while 13 reporters had between one and five years of experience. In contrast, 15 had between five and ten years of experience, while 13 reporters ($13/43 \approx 30.23\%$) had over ten years of experience. One person reported working on vulnerabilities for 21 years, another one reported 18 years; three more reported 15 years of experience.

Of the 27 participants of reporting SQL injection vulnerabilities, 25 responded to the question about their experience with a precise number of years. Most of them (21, $21/25 \approx 84\%$) had between one to ten years of experience. 4 reporters had more than ten years of experience.

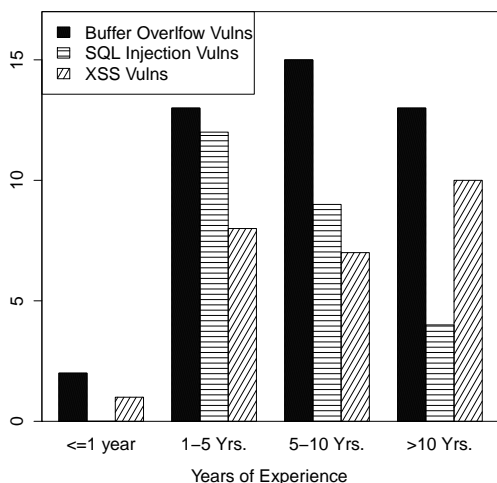


Figure 5.1: Experience of Reporters (Years Working On Vulnerabilities)

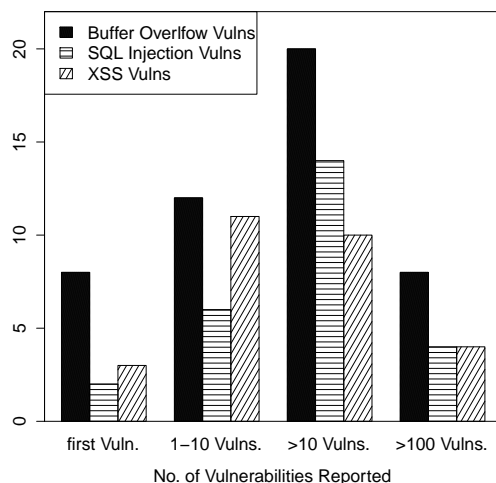


Figure 5.2: Experience of Reporters (Reported Vulnerabilities)

Among the 42 cross site scripting reporters, 26 responded to the question about their experience with a precise number of years. The group includes beginners as well as experts. Only 1 reporter had less than one year of experience. The rest of the reporters are almost evenly distributed among the other three experience brackets.

5.2 Single to a thousand vulnerabilities

Figure 5.2 shows the number of vulnerabilities reported by the study participants. 48 reporters of the buffer overflow study responded to the question about the number of vulnerabilities reported. A few participants are very experienced. We had 8 reporters who reported more than a hundred vulnerabilities. In fact, one of them publicly reported over thousand vulnerabilities. Most of them, 20 in total, reported more than ten vulnerabilities. 12 others said they had reported fewer than ten. Only 8 reporters said that the vulnerability in question was their first.

26 reporters responded to how many SQL injection vulnerabilities they reported. Only 2 reporters said that the vulnerabilities in question was their first; on the other hand, 4

reporters reported more than a hundred vulnerabilities. Most reporters reported more than ten, but less than one hundred vulnerabilities.

28 reporters answered the question about the number of cross site scripting vulnerabilities they reported. Most of them reported between one to ten vulnerabilities, but reporters with more than ten but less than one hundred reported vulnerabilities were equally common. 3 reporters said that the vulnerability in question was their first. 4 reporters reported more than a hundred cross site scripting vulnerabilities.

5.3 A blend of professionals and hobbyists

For all three studies, we had responses mostly from reporters who work as security professionals, but there are also a good number of reporters who work on detecting vulnerabilities as a hobby. For buffer overflow, we had 53 responses. Most of the reporters who responded were professionals working on security vulnerabilities—34 of them ($34/53 \approx 64.15\%$). Some of them are experts: 8 had over ten years of experience. 10 professional developers identified themselves as software developers who have security as a secondary interest. 9 identified themselves as hobbyists.

All of the 27 SQL injection vulnerability reporters responded about whether he/she considers vulnerability discovery as a hobby or as a profession. Similar to the trend in buffer overflow, most of the reporters were security professionals—18 of them ($18/27 \approx 66.67\%$). 9 reporters explored the vulnerability as a hobby. Most of these reporters (6) had less than five years of experience.

We had 38 responses from the cross site scripting reporters. Again, most reporters (25, $25/38 \approx 65.79\%$) identified themselves as security professionals. 8 other reporters worked as a software professional, but not on security. Only 5 reporters ($5/38 \approx 13.16\%$) identified themselves as hobbyists.

It is not surprising that we received responses from security professionals. People who start detecting vulnerabilities as a hobby end up as a professional working on security. Many

reporters stated this (“3 or 4 years as a hobby... It will soon be my professional responsibility as I have just signed a contract for a security company”, “It started as a hobby and now I am employed doing Security Consulting”). The hobbyists consider their effort as a learning process, so that they can become professional in future (“but hopefully one day I can get a permanent position working on Security Research / Penetration Testing.”). Some of these reporters get a material benefit for subsequent sale at repositories that pay back, e.g., ZDI [35] and iDefense [36].

5.4 Reporters explore different variants of vulnerabilities

We asked the reporters about the variant of vulnerability they explored, the platform they used, etc., in order to ensure that the group represents different perspectives. For buffer overflow, the 58 responders worked on 55 different software. Among these, 32 are Windows-based and 20 are Unix-based. 3 have both Windows and Unix versions.

Our responders have worked on operating systems such as Microsoft Windows and Linux; popular software such as Microsoft Visual Studio and Microsoft .NET framework; driver software such as the driver for BlazeVideo HDTV Player; and systems that protect critical infrastructure, e.g., Schneider Electric’s Interactive Graphical SCADA System, GE Energy D20/D200 Substation Controller, etc.

The 27 SQL injection reporters explored different kinds of injection scenarios. Most of the reporters (22) targeted the SELECT statement. Others targeted INSERT, UPDATE, and DELETE statements. 11 reporters ($11/27 \approx 40.7\%$) explored second order SQL injection attacks.

Cross site scripting is similar to other web vulnerabilities, e.g., cross site request forgery, HTML injection, etc. 16 reporters mentioned that they have experience with all kinds of web vulnerabilities. However, 24 reporters explicitly mentioned that they only have experience with detecting cross site scripting vulnerabilities.

Chapter 6

Study on Reporters of Buffer Overflow Vulnerabilities

In this section, we will describe the results collected from 58 reporters who reported buffer overflow vulnerabilities that were featured during the study periods. The results are described considering four groups of research questions (Section 4).

6.1 RQ1: General Approach

The first group of research questions focuses on the general approach, e.g., whether reporters actively search for vulnerabilities, how reporters choose target software, and whether they follow a common method.

6.1.1 Reporters start blindly

We distinguish between three levels of familiarity—a reporter of a vulnerability can (1) be a part of the software’s development team (*developer*), (2) select a software that he/she uses (*user*), or (3) select a software from some source to explore a vulnerability (*unfamiliar*). Surprisingly, a lot of the reporters (25, $25/58 \approx 43.10\%$) fell in the third category; *they decided to work on the target software just out of curiosity, boredom, or hunch*. One of them said, “I just downloaded an evaluation version on a rainy and boring day.” Another reporter said, “I have never used the program. I don’t even know exactly what it does ... The only thing in which I was interested was the fact that there were files with registered extension and so a real security scenario.”

A lot of reporters selected a target software from amongst the software used by them (27, $27/58 \approx 46.55\%$). But only a few reporters fell in the developer category (6, $6/58 \approx 10.34\%$). It is surprising that only a few reporters are affiliated with the development

How was the Vulnerability Discovered?				
Serendipitous Discovery	4	4	4	
	Explore For Vulnerabilities	21	23	2
		Unfamiliar	User of S/W	Developer
		Familiarity with Target Software (Buffer overflow vulnerabilities)		

Figure 6.1: Context of Discovery

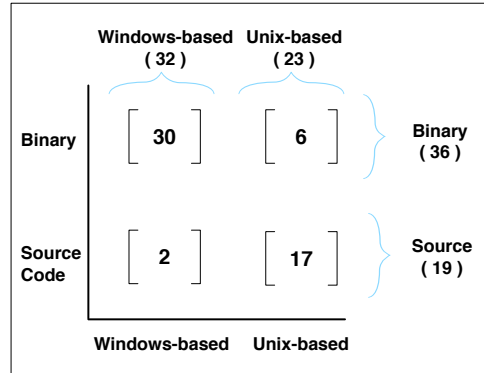


Figure 6.2: Platforms and Code

process. This is perhaps because the vulnerabilities discovered by developers are internally fixed as bugs; they are never reported to repositories.

6.1.2 Reporters actively probe popular targets

Figure 6.1 matches the reporters’ initial familiarity with a software with the circumstances of finding a vulnerability. It reveals several trends:

Vulnerabilities are not discovered serendipitously; reporters actively explore for them. Even most of the serendipitous discoveries are a result of reporters exploring for vulnerabilities in some other parts of the code. For example, 4 reporters serendipitously discovered the vulnerabilities when they were testing the software; 5 reporters found the vulnerabilities while doing code reviews (Figure 6.3). Sometimes, a ‘lucky’ runtime failure leads to a vulnerability, but they are less common (3 in Figure 6.3).

Reporters explore vulnerabilities in software that they download from the Internet or in software that they use. The target selection process is not entirely random. We asked reporters about how they select an application to target; 20 reporters responded. Most respond that their choice is influenced by *the popularity of target software* (11, 11/20 \approx 55%). Some reporters additionally mentioned that they target software that has *high visibility* (4, 4/20 \approx 20%), e.g., SCADA systems. Sometimes reporters have a *hunch about some software being easy targets* (5, 5/20 \approx 25%), such as a network service (“Network facing daemons are

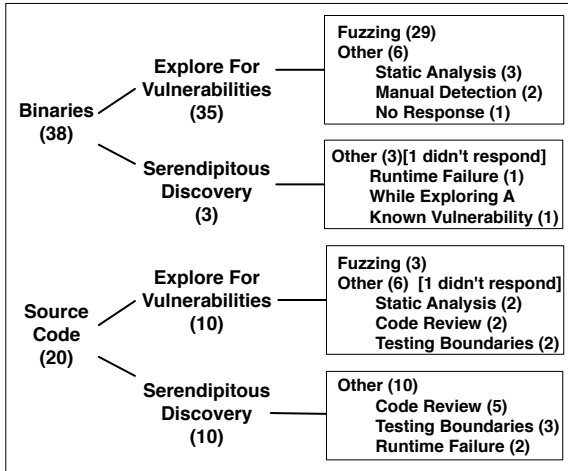


Figure 6.3: Popularity of Fuzzing

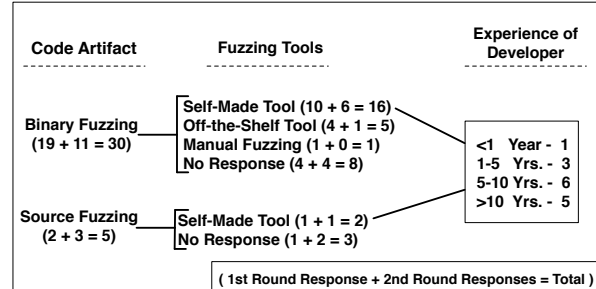


Figure 6.4: Custom Fuzzing Tools

desirable targets”) or a software written in a particular language (“I mainly choose software written in C / C++ as a target application. In this language there are many ways to make a mistake, and most of them are likely to happen if the programmer is not experienced enough.”). Only one person responded that the choice of the application was random.

6.1.3 Windows binaries and Unix source code

Of the 58 reporters, 55 respond to the questions about the code artifacts and the platforms they worked on.

Reporters exploring Windows-based software have to work mostly on binaries and reporters exploring Unix-based software have to work mostly on source code (Figure 6.2). Since developers generally adopt a fuzzing approach (Section 6.1.4), familiarity with the software or access to source code is not necessary. Many responses support this. For example, one reporter found a vulnerability in Adobe Flash on the Unix platform. He replied, “We do have access to the source code to some degree, as Adobe has open sourced its ActionScript virtual machine. However, because fuzz testing is a black box approach, we don’t need that kind of details to accomplish our task.”

6.1.4 Fuzzing is the chosen method

The responders unanimously agree that *fuzzing is the most suitable method for detecting a buffer overflow vulnerability* (Figure 6.3). Developers exploring binaries almost exclusively use fuzzing. Fuzzing is also used by reporters working on source code, but not as exclusively.

The reporters suggest that the general approach to explore for a buffer overflow vulnerability is to run a fuzzer and find a failure. Then one should use a debugger to trace through the fault and identify a buffer overflow situation. At this point, a vulnerability can be reported.

If a beginner is interested in creating an exploit, specifically on the Windows platform, (s)he can follow the recipe of one of the reporters: “1. Find a bug; 2. Examine the memory registers to see what it looks like; 3. See if it overwrites EIP (Extended Instruction Pointer) or SEH (Structured Exception Handler); 4. Determine how much space you have to work with. This will dictate if you have to use an egghunter or some sort of stack alignment technique; 5. Find any bad chars; 6. Tidy things up and complete the exploit”.

6.1.5 Applying static analysis is uncommon

Surprisingly, *reporters did not follow static analysis approaches to detect vulnerabilities*. Figure 6.3 shows that only 5 reporters used static analysis. 2 other reporters said that they did not use static analysis for the specific case, but they used static analysis for detecting other buffer overflows.

There seems to be a general consensus about adopting fuzzing. One reporter who applied static analysis on binaries even admitted, “I’m one of the few that rarely fuzz”.

It may happen that developers use static analysis during writing and testing their code and they were not represented in the study. Indeed, developers are a minority in our study—only 6 reporters were directly involved with developing the vulnerable software that were reported (Figure 6.1). However, only one of them used static analysis. In contrast, 3 developers discovered overflows during code review.

6.1.6 Code review and other manual approaches are useful in detecting vulnerabilities in source code

Reporters working on source code often detected vulnerabilities manually. Among the 20 reporters working on source code, 7 detected overflows during code reviews (Figure 6.3). Only one of the vulnerabilities—a buffer overflow in the Linux kernel—was detected through a systematic code review process. Two vulnerabilities were detected when the reporters were working on fixes for other bugs in the same code; both of these reporters were developers of the software. In the remaining 4 cases, reporters were trying to understand third-party code when they found the vulnerabilities.

In 5 instances, reporters detected buffer overflow vulnerabilities by trying to systematically test the target software (Figure 6.3). 3 reporters tested the target software based on an assumption that a critical feature may harbor vulnerabilities (serendipitous discovery). For example, one reporter said, “I was looking at how HTTP headers were added to a string and started to wonder how the boundary checks worked... I had seen a similar vulnerability reported against the project a few months prior, which made me wonder whether there were other unchecked strings.” In 2 other cases, reporters were exploring the software to detect buffer overflows, but they did not use fuzzing; instead they wrote test cases to simulate the bug using debuggers in the process.

6.2 RQ2: Use of Tools

Following the general method discussed in Section 6.1.4, fuzzing and debugging tools are primarily used by reporters. In this section, we report about whether reporters use off-the-shelf tools and whether they use a few common tools.

6.2.1 Off-the-shelf fuzzing tools are typically not used

Although fuzzing tools are more common, *people typically do not use off-the-shelf tools; they prefer making their own fuzzers* (Figure 6.4). These fuzzers target various protocols

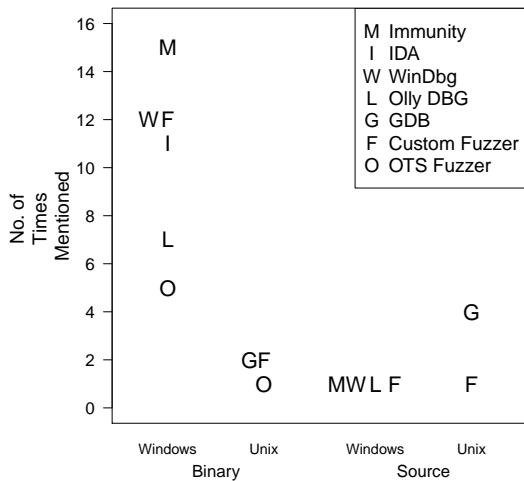


Figure 6.5: Tools In Use

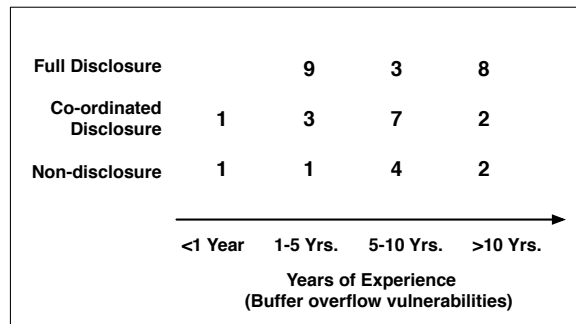


Figure 6.6: Reporting Practices

(SSH, IMAP, etc.). Other tools target complex data types. For example, one reporter wrote a fuzzing tool targeting programs that take as input textual data that is regulated by a grammar, such as regular expressions. Interestingly, this is the only tool written by a researcher; it was written for a research work that was published at a security conference.

Both new and expert reporters build fuzzing tools; but it is more common for experts to build such tools (Figure 6.4). For new reporters, the experience is a part of the learning process (“I wanted to see if I could do it , that’s the only reason”, “Although ‘reinventing the wheel’ is not everyone’s thing, I feel it helps me understand how tools and the protocols truly work”).

18 reporters developed custom fuzzing tools. Most (14) of the custom tools were ad hoc; they were used once. Three reporters mentioned that they reuse their fuzzing tools; however, they added that the tools were developed in their respective organizations. Their original intent was to find good case studies for their tools. One other reporter, who works at Google, used an internally developed fuzzer.

Among the off-the-shelf fuzzing tools, reporters mostly mentioned open source tools (SPIKE, COMRaider, Radamsa, Sulley, and Peach). One reporter said that “off-the-shelf

tools typically cost money and independent researchers strive for higher ROI, especially when resources are low.”.

6.2.2 Immunity Debugger, IDA Pro, and WinDbg in Windows platform, GDB in Unix platform

Reporters mentioned 51 different debuggers; Figure 6.5 shows the debuggers and other tools mentioned. *Among Windows-based debuggers, Immunity Debugger, IDA Pro, and WinDbg are the most common.* Immunity Debugger, a powerful tool for writing exploits, is mentioned the most times as a debugger (mentioned 15 times). One of its extension packages written in Python, named Mona, is mentioned 4 times. IDA Pro, a Windows-based commercial tool, is mentioned but mostly as a tool for exploit generation. *Among Unix-based tools, GDB is the chosen debugger* (mentioned 6 times).

Different reasons are given by reporters about why they use Immunity Debugger; we had 7 reporters commenting on this. All of them suggested that Immunity Debugger is very easy to use. Four developers mentioned that Immunity Debugger is the most useful because of the Mona plugin (`mona.py`)¹. Other responses suggest that IDA Pro and WinDbg are perhaps more versatile and used for more complex jobs; however they are more difficult to use (“For more complex reversing jobs, I’m using IDA.”, “I SHOULD be using WinDbg, that is the most powerful debugger, but it is also the hardest to use.”). Most reporters in the study who used IDA Pro and WinDbg were security professionals with over 5 years of experience. On the other hand, the experience of users of Immunity Debugger ranged from one year to more than 10 years. Thus the reasoning that Immunity Debugger is easier to use has some truth in it.

¹From Jan 2013, Mona is also available with WinDbg.

6.2.3 Static analysis tools are almost never used

Reporters prefer fuzzing-based blind search over more systematic white box exploration methods. That is why static analysis tools have limited applicability during vulnerability detection (Figure 6.3). One reporter mentioned Coverity Security Advisor tool (commercial), but not as a tool he used to detect the vulnerability in the study. Instead, he mentioned that he had used Coverity for detecting other vulnerabilities. Another reporter mentioned a custom tool called Kint, while a third reporter mentioned Flawfinder, an open source tool. One other reporter mentioned that he uses static analysis for development work, but uses fuzzing for detecting vulnerabilities.

11 reporters mentioned using IDA Pro (Figure 6.5). IDA Pro has multiple features—it is a disassembler with various static analysis, a powerful debugger, and a decompiler. However, all the reporters mentioned that they used IDA Pro as a debugger. A few mentioned using the WinDbg plugin available in IDA Pro, because they were familiar with IDA Pro, but also wanted WindDbg features.

6.3 RQ3: Vulnerability Reporting

This section explores how and where do reporters report, and what are the consequences of their decisions. We distinguish three reporting practices: (1) *non-disclosure* (contact with vendor only), (2) *coordinated disclosure* (contact with vendor first, then make public), and (3) *full disclosure* (publicly release vulnerability without contacting the vendor).

6.3.1 A lot of reporters prefer full disclosure

54 reporters answered about their reporting activity.

Many reporters adopt the full disclosure approach and post to vulnerability lists (28, 28/54 \approx 51.85%). Other reporters communicate with vendors: some release the vulnerabilities later in a mailing list (co-ordinated disclosure), others do not (non-disclosure). The study

results in both rounds suggest that coordinated disclosure (16, 29.63%) is a little more popular than non-disclosure (10, 18.52%).

Figure 6.6 shows the reporting preference in terms of the experience of the reporters. There is no clear trend here, but it shows that full disclosure is preferred by all kinds of reporters. However, reporters do show some concern about the consequences of full disclosure. Some suggest that they contacted the vendors, but the vendors never replied.

6.3.2 Experts choose highly visible reporting options

We have 13 reporters who have more than ten years of experience, of whom 12 answered the question about reporting. We wanted to track how they reported vulnerabilities.

These experts typically choose reporting options with high visibility. The first reporter mentioned that he used to keep track of Bugtraq, but is “no longer doing it, mainly for lack of interest.” Instead, he uses Twitter to disclose vulnerabilities. The second reporter said that he does not check forums anymore. Two other reporters use Twitter, including one who used to work as a manager at a well-known vulnerability repository. The fifth reporter said that for the particular vulnerability he reported, he directly contacted the security lead. Another reporter contacted Microsoft to report a bug, because it provided high visibility. The seventh one prefers reporting through ZDI for monetary incentives. The eighth one disclosed the vulnerability, which is about a SCADA system, at a security conference to get public attention (“I chose this forum because the vendor showed no interest in fixing security problems. I wanted to pressure the vendor ... via press exposure.”). The ninth person only reported to a repository established and maintained by him. Two other reporters were developers of the target applications.

Only one reported to a vulnerability repository. This reporter is a software professional (Section 5.3), not a security professional; this may explain the difference in approach.

6.3.3 Fully disclosed vulnerabilities may remain unfixed

Experienced reporters justify full disclosure as a way to put pressure on vendors to fix vulnerabilities. They expect vendors to pay attention to their activity (“now I use Twitter when I like to show the releasing of a vulnerability that deserves some attention.”). In practice, it may not work.

Surprisingly, *if reporters, both beginners and experts, disclose publicly but do not contact vendors, the vulnerabilities are likely to remain unfixed*. Figure 6.8 shows that 20 of the 27 publicly disclosed vulnerabilities remain unfixed (74.07%). We determined whether a vulnerability remains unfixed by searching for patch information at multiple sources: SecurityFocus, NVD, Secunia, OSVDB repositories, vendor webpages, etc. Each of the repositories periodically update by collecting information from the vendors and other repositories. Hence, if a vulnerability gets fixed, the information is likely to be available from these sources.

Even if the reported vulnerabilities have detailed exploit code, most remain unfixed (13 out of 14 in Figure 6.8). To date (March 24, 2014), these 13 vulnerabilities have remained unfixed for an average of 774 days since they were reported. Some of these are important software, such as a Windows graphics driver of nVIDIA (unfixed 260 days), or a SCADA system controller of GE (522 days). There are five CVEs associated with these 13 vulnerabilities; the average Common Vulnerability Scoring System (CVSS) scores [23] for these vulnerabilities are high (9.3 out of 10) with high average impact sub-score (10 out of 10) and high average exploitability sub-score (8.6 out of 10). Thus leaving the vulnerabilities unfixed may have severe impact. The only fixed vulnerability was fixed by Microsoft.

6.3.4 Exploit Database (EDB) is the most popular

Reporters mention 55 repositories in their answers.

Most favor Exploit Database (EDB); it is mentioned 16 times (29.09%). All reports to EDB had some exploit code; 12 had detailed exploit code (Section 6.4). Other repositories

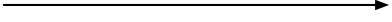
Detailed Exploit	1	10	5	5
DoS Exploit	1	1	7	7
No Exploit		1	5	1
				
	<1 Year	1-5 Yrs.	5-10 Yrs.	>10 Yrs.
	Years of Experience (Buffer overflow vulnerabilities)			

Figure 6.7: Exploit Generation

Exploit Generation					
	None	3	4	13	4
Detailed Exploit		3	4	13	4
DoS Exploit	4	8	6	6	4
No Exploit	6	1	1	1	2
	Non-disclosure	Co-ordinated Disclosure	Full Disclosure		
	Reporting Practices (Buffer overflow vulnerabilities)				

Unfixed

Fixed

Figure 6.8: Which Problems Are Fixed?

such as SecurityFocus (11 times), Secunia (6 times), OSS-Security (6 times), and CERT (4 times) are mentioned. 5 reporters posted to ZDI, because of commercial interest.

6.4 RQ4: Exploit Generation

This section explores whether reporters value the effort on writing exploits. We distinguish three kinds of exploits: (1) no exploit code written, (2) a simple denial-of-service (DoS) exploit that crashes the vulnerable application, and (3) a detailed exploit that takes control of the computer system.

6.4.1 Reporters typically write exploits

We have 57 reporters answering questions about exploits. 47 of them ($47/57 \approx 82.46\%$) prepared some exploit code to the vulnerability report. Half of these reporters prepared exploits that only crashed the application (24, $24/57 \approx 42.1\%$ overall). Other reporters created detailed exploits that executed the reporters code (23, $23/57 \approx 40.35\%$ overall). 10 reporters did not prepare any exploit code (17.54%).

6.4.2 Perception of exploit generation effort varies

Reporters, as they get more experienced, think that their time is more well-spent in finding new vulnerabilities as opposed to writing detailed exploit codes (“I’m not interested in

demonstrating code execution at 100% because my hobby is just finding the vulnerability.”—Reporter with 12 years experience). They often write exploits only to demonstrate the crashes (DoS exploit, Figure 6.7). On the other hand, beginners take exploit generation as a challenge by itself, which, they believe, will enhance their knowledge. These professionals regard exploit generation with high esteem. They were the more enthusiastic bunch; their responses were very detailed especially in the part where they answered how they generated the exploits. Other new reporters aspire to become an exploit developer. For example, a reporter who had three years of experience but reported only one vulnerability replied about why he did not write an exploit code, “I’m not competent enough to do so.”

We investigated the 5 reporters with over ten years of experience who wrote exploits (Figure 6.7). Two mentioned doing it as their job responsibility. One reporter developed an exploit because the target was highly visible. Another reporter demonstrated the exploit in a security conference. Therefore he spent the time to develop it; otherwise he would not have developed a detailed exploit (“it was my first Metasploit module”). The fifth reporter developed the exploit in 30 minutes only, perhaps reusing some existing code (since detailed exploit development takes time). Thus these reporters had some incentives in developing detailed exploits.

6.4.3 Exploits are written to make public

Figure 6.8 matches the exploit generation and reporting practices of reporters. It shows that reporters who spent time on exploit development eventually released the exploit code, either independently or after co-ordinating with vendors. In fact, when reporters wrote detailed exploits, they always released exploit code.

Conversely, full disclosure almost always has some exploit code—minimal or detailed. When a reporter reports to a vendor only, his/her reports may contain a minimal exploit, but not a detailed exploit.

Chapter 7

Study on Reporters of SQL Injection Vulnerabilities

We received 27 responses from reporters of SQL injection vulnerabilities. This is lower than the responder numbers of the other two vulnerabilities, partly because we started with fewer reporters (Table 3.1). This section presents the analysis results.

7.1 RQ1: General Approach

We first focus on the general approach of the reporters, specifically how they select a target to explore and what methodology they follow.

7.1.1 Reporters explore for bugs in software that they use

We explained earlier three levels of familiarity—a reporter of a vulnerabilities can (1) be a part of the softwares development team (*developer*), (2) select a software that he/she uses (*user*), or (3) select a software from some source to explore a vulnerability (*unfamiliar*).

None of the reporters identified themselves as developers. Most (15, $15/27 \approx 55.56\%$) said that they target a software that they use. The remaining 12 reporters fell in the third category—they targeted an application that they had never used before.

The fact that we did not have any developers is not surprising. Even for buffer overflow, there were only 6 reporters who identified them as developers. Most of the vulnerabilities detected by developers are fixed during the development phase. Only a few make their way to a vulnerability repository.

Figure 7.1 matches the reporters' initial familiarity with a software with the circumstances of finding a vulnerability.

Vulnerabilities are not discovered serendipitously; reporters actively explore for them.

There are only 3 reporters (3/27 \approx 11.11%) detected the vulnerability serendipitously. All of these discoveries are a result of reporters exploring for vulnerabilities in some other parts of the code. For example, one reporter was exploring commits made to a repository when he found a bug in committed code. He mentioned that attackers often watch commits made to repositories in a similar way. Although he categorized the discovery as serendipitous, he added that it may be an actual vulnerability exploration process in another context.

A buffer overflow can be triggered suddenly by a user’s action. The situation is unlikely for an SQL injection vulnerability. This explains why the vulnerabilities are almost always found by reporters actively exploring.

7.1.2 Reporters target applications for specific reasons

Reporters follow various methods to target the software they explore. Although 8 reporters said that the target selection process was entirely random, others came up with specific reasons.

8 reporters said that their choice is influenced by the *popularity of target software or the high visibility* (“The application was in scope because it is a product of a famous vendor and is in use by big companies...”, “I was working at ... and this software is controlling most of the GRID infrastructure”). 7 reporters mentioned that they target a class of software—content management systems—that they think are vulnerable as a whole (“I’ve picked top couple of hundred (cca. 200-300) WordPress plugins and searched for vulnerabilities inside.”, “apps with high possibility of user interaction is always a safe bet”). 5 mentioned that they only select software that match their skill set (programming language, database type, etc.).

7.1.3 Most reporters explore for SQL injection vulnerabilities manually

Two third of the reporters (18 out of 27) said that they detected the vulnerability manually without the help of any detection tools. A few people mentioned that they only

How was the Vulnerability Discovered?			
	Unfamiliar	User of S/W	Developer
Serendipitous Discovery	1	2	0
Explore For Vulnerabilities	11	13	0

Familiarity with Target Software (SQL injection vulnerabilities)

Figure 7.1: Context of Discovery

Full Disclosure	10	7	1
Co-ordinated Disclosure	2	1	1
Non-disclosure		0	2

<1 Year 1-5 Yrs. 5-10 Yrs. >10 Yrs.

Years of Experience (SQL injection vulnerabilities)

Figure 7.2: Reporting Practices

used an intercepting proxy. When we matched the data with the reporters who had access to source code (20 of them), we found that 16 of them were following the manual approach. This is perhaps because SQL injection happens at database access points. A simple search can find the places in a program that accesses data. A reporter can then detect a vulnerability by analyzing the program context around each database access points. This approach was mentioned by a reporter as an algorithm: “(1) Install the app on my local system; (2) Read any documentation (if existing); (3) Fire up the application and have a look around trying to get a view of what’s going on; (4) Open the source in my favorite editor and take a look at how it’s laid out; (5) Start using grep/find commands to search for possible ‘injection’ points (Here it’s important to understand how the app works, since apps for WordPress and MyBB may handle user requests differently); (6) Start testing any possible vulnerability against the app; (7) Repeat 4-6 until I either fail or succeed”

7 reporters did not have access to the source code. So they were treating the software as a black box. Such reporters are more likely to use tools; we found 5 out of these 7 reporters used penetration testing tools. 2 reporters said that they still followed a purely manual, trial and error based process.

7.2 RQ2: Use of Tools

7.2.1 Reporters mention using penetration testing tools

12 out of 27 reporters mentioned that they have used some automated tools to detect SQL injection vulnerabilities. Some of them manually detected the specific vulnerability we asked about, but they said they had used tools for other detection scenarios.

Reporters mainly named two tools: sqlmap (5 reporters) and Burp Suite (6 reporters). sqlmap is an open source penetration testing tool that supports testing many databases and operating systems for different variants of SQL injection vulnerabilities. Burp Suite is mainly an interception proxy that lets a user inspect and modify traffic between a browser and the target application. However, it has several other features—even an automatic scanner to detect various types of web vulnerabilities, e.g., SQL injection and cross site scripting. Two reporters mentioned that they used Burp Suite only as an interception proxy, but others said that they have used the scanning feature.

7.2.2 Static analysis tools are almost never used

As mentioned in Section 7.2.1, 12 out of 27 reporters mentioned using some tools. Reporters did not follow static analysis approaches to detect vulnerabilities, even though they had access to the source code in many situations. Only one reporter mentioned using a static analysis tool, but the reporter was an academic testing his SQL injection vulnerability detection mechanism on open source software.

Only one reporter explained why he did not use a static analysis tool. He said that the available static analysis tools do not meet his needs (“Main reason for that is because they can’t handle OOP PHP and most CMS’s does OOP”).



Figure 7.3: Exploit Generation

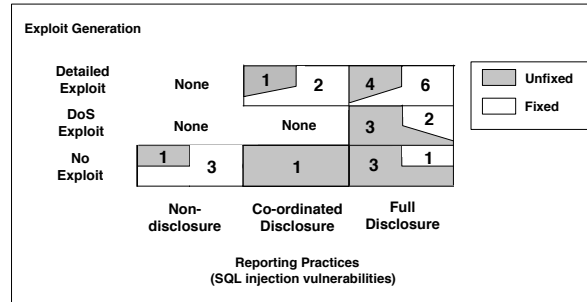


Figure 7.4: Which Problems Are Fixed?

7.3 RQ3: Vulnerability Reporting

7.3.1 A lot of reporters prefer full disclosure

All 27 reporters answered about their approaches to report vulnerabilities.

Most reporters post the vulnerabilities directly to vulnerabilities lists (19, 19/27 \approx 70.37%), i.e., prefer full disclosure. 8 other reporters contacted vendors. 4 of them later posted the vulnerabilities to vulnerability repositories, i.e., co-ordinated disclosure. 4 others only reported to vendors.

Although reporters disclosed vulnerabilities publicly, many expressed concerns about the consequences of full disclosure. Many said that they contacted the vendors, but the vendors never replied (“I report vulnerabilities directly to the software project, if that fails I choose an open disclosure method fitting to the vulnerability found”).

Figure 7.2 cross references the experience of the reporters with their reporting practices. It shows that experts show restraint about reporting SQL injection vulnerabilities in public, but it is strongly favored by new reporters.

7.3.2 Fully disclosed vulnerabilities may remain unfixed

Similar to the buffer overflow study (Section 6.3.3), we further explored the reported vulnerabilities and identified how many of them got fixed. Figure 7.4 shows this. It shows

that 10 of the 19 fully disclosed vulnerabilities remained unfixed. Of them, 4 were reported with detailed exploit code.

As of October 23, 2014, these 10 vulnerabilities have remained unfixed for an average of 643 days since they were published. Some of them are vulnerabilities from popular software, such as a Windows graphics driver of Joomla! (unfixed 679 days), MyBB (unfixed 719 days), or WordPress (average 805 days).

7.3.3 Exploit Database (EDB) is the most popular

We also analyzed which vulnerability repository is most popular for detectors to report vulnerabilities and why they were selected.

Exploit Database (EDB) was mentioned 10 times. Reporters consider it to be highly visible by the vendors and other security communities, hence giving them the most exposure (“I used it because all the other outlets will pick it up from there”). Also, EDB has formed an active and loyal community (“...because I enjoy giving back to those which provide valuable resources and time for free”). All reports to EDB had some exploit code; 5 reporters posted detailed exploit code. Some other popular repositories were also mentioned, e.g., Security-Focus was mentioned 3 times, Secunia was mentioned 2 times, and Bugtraq was mentioned 3 times.

7.4 RQ4: Exploit Generation

As mentioned before, we distinguish three kinds of exploits: (1) no exploit code written, (2) a simple denial-of-service (DoS) exploit that crashes the vulnerable application, and (3) a detailed exploit that takes control of the computer system.

7.4.1 Reporters typically write exploits

All 27 reporters answered the question about whether they wrote exploit code or not.

22 of them (22/27 \approx 81.48%) prepared some exploit code. Most of them (17, 17/22 \approx 77.27%) wrote detailed exploit code. Others (5 reporters) were interested to show “some impact through the exploit of the vulnerability”, but not detailed exploit code. Another 5 reporters did not prepare any exploit code. Most said that SQL injection vulnerabilities were not complex, so the effort of preparing a detailed exploit was not intellectually challenging (“I did not publish any PoC exploit because it’s a lot easier to exploit the vulnerability through a HTTP Request handler (BURP, Live HTTP Headers)”). Even people who created detailed exploits agreed that recreating a SQL injection vulnerability is not that complex and it was relatively easy for them to develop the exploit code, specifically using available tools (“by using sqlmap, it took less than 5 minutes”).

7.4.2 Exploits are written to make public

When we compared the data with the experience of the reporters, we found that experts as well as beginners had shown preference to write detailed exploit code (Figure 7.3). Among 3 experts (more than 10 years of experience) who made detailed exploits, 2 of them reported publicly. Similarly, 5 reporters out of 7, who had less than 5 years of experience, reported the vulnerabilities with exploit code publicly.

Chapter 8

Study on Reporters of Cross Site Scripting Vulnerabilities

The analysis results in this section were derived from 42 cross site scripting reporters who responded to our study.

8.1 RQ1: General Approach

8.1.1 Reporters explore for bugs in software that they use

We asked if a reporter was involved in the development process of the target software (*developer*), only a user (*user*), or unfamiliar with the software that he chose to explore (*unfamiliar*). 41 reporters responded to this question.

Most of the reporters (19, $10/41 \approx 24.39\%$) said that they selected a target software from amongst the software they used. Interestingly, we received responses from 9 reporters who identified them as developers, either directly involved in the process (“I used to be the tech lead of this application, before moving to a security job. I was also part of the security team of the application.”) or an active maintainer (“I have tried to help maintain the LQT extension a bit, so I would say closer to developer than end user (I am not the main author however.)”).

Only a few reporters (10, $10/41 \approx 24.39\%$) said that they were unfamiliar with the target software.

Figure 8.1 shows that reporters actively explore for cross site scripting vulnerabilities. Even some of the serendipitous discoveries are a result of reporters exploring for vulnerabilities in some other parts of the code. One reporter said that he, “...was looking at proof-of-work schemes (timelock, hashcash), the XSS vulnerability was a chance find by a

How was the Vulnerability Discovered?			
Serendipitous Discovery	2	1	5
	Explore For Vulnerabilities	8	18
	Unfamiliar	User of S/W	Developer
	Familiarity with Target Software (XSS vulnerabilities)		

Figure 8.1: Context of Discovery

	Years of Experience (XSS vulnerabilities)			
	<1 Year	1-5 Yrs.	5-10 Yrs.	>10 Yrs.
Full Disclosure	1	2		3
Co-ordinated Disclosure		3	3	2
Non-disclosure		3	4	2

Figure 8.2: Reporting Practices

global cursory code review of the module”. Another reporter said, “it was really just a coincidence as I was looking for issues in the application in general and this code just happened to be deployed with the code we were given to review”.

8.1.2 Reporters stated various reasons about why they target an application

As Figure 8.1 suggests, most of the reporters detected vulnerabilities in applications used by them. 12 reporters said that they were testing the target application as a professional responsibility.

Among the reporter who targeted random applications, only 2 said that they target applications with high visibility. One mentioned he targeted an application because the vendor was offering a reward.

8.1.3 Most reporters explore for cross site scripting vulnerabilities manually

Most of the reporters (34, $34/42 \approx 80.95\%$) said that they preferred detecting the cross site scripting vulnerabilities manually.

We matched the data with whether reporters had access to source code or not. Of the 42 reporters, 40 responded to this question. 29 reporters ($29/40 \approx 72.5\%$) had access to source code. Of these reporters, 25 said that they manually explored for the vulnerability.

8 reporters did not have access to source code, yet they followed a trial and error based manual approach. Manual exploration is feasible since a reporter only has to focus on the parameters of the URL where any text input can be fetched. A starting point may be the input boxes of an application. File upload can also be another target, as a reporter can rename a file with the XSS script and try uploading it. Since a cross site scripting vulnerability can be detected following these simple tasks, automated tools are not appealing.

8.2 RQ2: Use of Tools

Following the discussion of the general methodology followed by reporters, most prefer manual detection. However, they mentioned a few tools that they use. This section reports that.

8.2.1 Reporters mention using penetration testing tools

8 reporters (8/42 \approx 19.05%) said that they detected the vulnerability using tools. Most reporters mentioned Burp Suite, which is a web interception proxy that also has many other features attractive for a web penetration tester. However, it was not clear from answers whether reporters used Burp Suite only as an interception proxy, or use its scanner features. Many reporters praised that Burp Suite is the best tool for analyzing the HTTP request and response, and it allows reporters to bypass client side limitations such as field size and JavaScript.

Reporters mainly named two tools: sqlmap (5 reporters) and Burp Suite (6 reporters). sqlmap is an open source penetration testing tool that supports testing many databases and operating systems for different variants of SQL injection vulnerabilities. Burp Suite is mainly an interception proxy that lets a user inspect and modify traffic between a browser and the target application. However, it has several other features—even an automatic scanner to detect various types of web vulnerabilities, e.g., SQL injection and cross site scripting. Two

reporters mentioned that they used Burp Suite only as an interception proxy, but others said that they have used the scanning feature.

8.2.2 Static analysis tools are almost never used

Surprisingly, reporters did not follow static analysis approaches to detect vulnerabilities. Only one reporter replied that he followed a static analysis approach, but even that was a lexical analysis tool. The tool is proprietary.

8.3 RQ3: Vulnerability Reporting

8.3.1 Reporters prefer to report to vendors

41 reporters answered about their reporting activity.

A lot of reporters ($21/41 \approx 51.22\%$) contacted vendors directly and did not post to a vulnerability repository, i.e., followed non-disclosure approach. 12 others reported to vendors first and then posted to a vulnerability list, i.e., followed co-ordinated disclosure approach. Only 8 reporters reported vulnerabilities publicly.

Figure 8.2 shows beginners as well as experts preferred communicating with vendors rather than reporting publicly.

Since vulnerabilities are directly reported to vendors they are fixed most of the time (Figure 8.4). Although the count is small, it shows that fully disclosed vulnerabilities were less likely to be fixed.

8.3.2 No specific repository is popular

27 reporters answered about where to report vulnerabilities. No specific forum was mentioned by more than four reporters. Drupal Security was mentioned 4 times, Secunia was mentioned 3 times, and Bugtraq was mentioned 3 times.

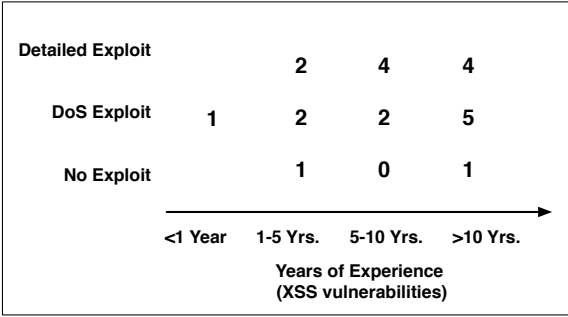


Figure 8.3: Exploit Generation

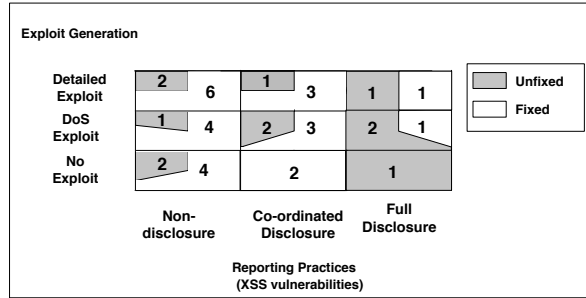


Figure 8.4: Which Problems Are Fixed?

8.4 RQ4: Exploit Generation

8.4.1 Reporters typically write exploits

36 reporters responded about making exploits.

Most of the reporters (26, $26/36 \approx 72.22\%$) prepared some exploit code—half of them prepared simple exploits that only crashed the application, but while the other half created detailed exploits.

Since cross site scripting vulnerabilities are relatively easy to spot, reporters said that they wanted to confirm the exploitability rather than just reporting the vulnerability as a theoretical issue. Also, typically it only takes a little time commitment to write an exploit, since the process is straightforward.

10 reporters (27.78%) did not prepare any exploit code. They argued that the exploit preparation is so easy that anybody can do that from a vulnerability report; hence there is no need to spend more time generating exploit.

Figure 8.3 compares the exploit generation data with the experience of the reporters. This does not show any trend except for the fact that exploit generation is popular among beginners as well as experts.

Chapter 9

Communities Formed Around Vulnerabilities

Each kind of security vulnerability is different. This impacts the methodologies and tools, and gradually the people. Thus different kinds of vulnerabilities form their own communities. Figure 9.1 shows evidence of such communities forming around the three target vulnerabilities. Of all the reporters who were featured in our study periods, only 2 reporters reported all three kinds of vulnerabilities. SQL injection vulnerabilities are often detected on web applications; this may justify the that there were a lot of reporters who concentrated on these two vulnerabilities. On the other hand, buffer overflow is very different from the other two, hence there were only a few reporters who reported buffer overflow and one of the other two vulnerabilities.

This section will compare the data of three kinds of vulnerabilities from different perspectives. We will specifically focus on how reporters select the target applications, whether they use tools or not to detect vulnerabilities, and how they report vulnerabilities.

9.1 Different reasons to select a target application

Both buffer overflow and SQL injection reporters said that the popularity of a software and the potential visibility is an important criteria that they consider while selecting it as the target. On the other hand, most of the cross site scripting vulnerabilities reporters detected the vulnerabilities from applications that they had used.

An explanation may be in the effort needed to detect a vulnerability. Intuitively, detecting a buffer overflow vulnerability is significantly harder than detecting a SQL injection vulnerability, while detecting a cross site scripting vulnerability is relatively easier. This is further supported by the empirical data that we collected about the effort needed to write

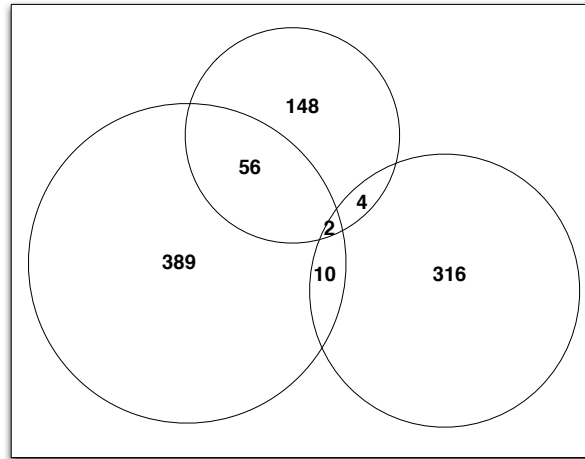


Figure 9.1: Set of Reporters Targeting Different Vulnerabilities

a detailed exploit. A reporter could generate a cross site scripting exploit code in minutes, whereas it took a reporter hours, even days, to generate a buffer overflow exploit. Therefore, reporters may try to optimize the recognition received from the effort they spent.

Reporters thought that detecting a cross site scripting vulnerability is not a challenging task; several reporters supported this. Hence there may be a lack of interest among reporters to actively select target software. Most of the vulnerabilities were detected by users of software and the developers themselves.

9.2 Buffer overflow reporters use tools, other reporters detect manually

Most of the buffer overflow reporters used fuzzing and debugging tools—almost always when they explored for vulnerabilities (Figure 6.3). On the contrary, reporters of SQL injection and cross site scripting mostly said that they detected the vulnerabilities manually. A few SQL injection reporters mentioned some tools, but cross site scripting reporters almost always follows a manual approach.

Manual approach is sufficient for efficiently detecting a SQL injection or a cross site scripting vulnerability, but not for a buffer overflow vulnerability. For a buffer overflow reporter, it apparently pays off to develop a custom fuzzing tool when automated tools

	Source Code	Binary
Buffer Overflow	19	36
SQL Injection	12	15
Cross Site Scripting	29	11

Table 9.1: Reporters With Access To Source Code

are not available. This is because a buffer overflow may be triggered by complex memory operations which are hard to detect on source code—harder on binary code. The error at a program point that triggers a buffer overflow may be different from the program point where the overflow is manifested. On the contrary, both SQL injection and cross site scripting happens at specific program points that can be found by simple `grep` operations.

Another possible explanation is that a manual approach does not work well for binary code. Table 9.1 shows that most of the buffer overflow reporters did not have access to the source code, i.e., they worked on binaries. In proportion, more SQL injection and cross site scripting reporters had access to source code. Hence, manual approach worked for them.

Overall, there are some good tools that are used by different reporters. For buffer overflow, reporters frequently use Immunity Debugger, IDA Pro, WinDbg, GDB, and other debuggers, Metasploit framework for exploit development. For SQL injection, many reporters mentioned sqlmap and Burp Suite. Burp Suite is also popular as an intercepting proxy among cross site scripting reporters. There remains a need for developing efficient, versatile, and usable tools for reporters. When such tools are available, developers adopt them to automate their manual approach.

	Non-disclosure	Co-ordinated disclosure	Full disclosure
Buffer Overflow	10	16	28
SQL Injection	4	4	19
Cross Site Scripting	21	12	8

Table 9.2: Vulnerability Reporting Practices

9.3 Cross site scripting reporters reported to vendors more

As shown in Table 9.2, cross site scripting reporters more often communicated with the vendors and did not publish the vulnerabilities in vulnerability repositories. On the other hand, buffer overflow and SQL injection reporters reported publicly most of the time.

Detecting a cross site scripting vulnerability is relatively easy. Also, it is well known that a lot of web applications have cross site scripting vulnerabilities, but they are not fixed since the impact of the vulnerabilities is benign or the web applications are not actively maintained. All of these may explain why detecting a cross site scripting is not as rewarding as detecting a buffer overflow. This also explains why people actively download random software and look for buffer overflow vulnerabilities, and why the same does not happen for cross site scripting vulnerabilities (Section 9.1). Moreover, we found that many cross site scripting reporters were developers themselves.

Chapter 10

Discussion and Recommendations

Our study had two significant results: (1) We found that reporters mostly use fuzzing and penetration testing, even follow manual approaches, despite the fact that researchers and tool vendors have concentrated on static analysis approaches. (2) We also found a problem in the way vulnerabilities are reported: reporters often do not coordinate with vendors and their choice to publicly disclose vulnerabilities, often with detailed exploits, is not useful for vendors to react and produce patches in time. We first discuss these issues.

Fuzzing and other approaches vs static analysis—Understanding the divide. Security vulnerabilities are not only detected by independent professionals and hobbyists (we refer to them as ‘non-developer reporters’) who are interested in finding flaws in random software, but are also detected by the developers who are directly affiliated with the software. Their goals are different. Non-developer reporters are interested in quickly finding isolated flaws in target software; on the other hand, developers are interested in finding and fixing all flaws in the code they write. This is why their choices differ.

Most of our study participants of buffer overflow and SQL injection study were non-developer reporters (Sections 6.1.1 and 7.1.1). They favor fuzzing and penetration testing approaches since it is easy to adopt and it quickly finds bugs (“a quick fuzzing test was enough to crash the application”). Also, the vulnerability detected is instantly reproducible. Reporters may spend more time on exploits or quickly move on to the next target. Since non-developer reporters want to maximize their effort, they prefer a quick approach, as long as it is effective. Fuzzing suits their needs perfectly. Even manual detection suits for detecting SQL injection and cross site scripting vulnerabilities (Section 9.2).

However, there are detection tools based on static analysis that are widely used during software development [7]. Perhaps, the developers are not well-represented in our study. However, even among the few developers, static analysis approaches were not mentioned much (Section 6.1.5). Also, there were more developers in the cross site scripting study, but even they used manual detection approaches, e.g., code review. It may also happen that developers relate static analysis with coding and testing and not necessarily with security analysis and vulnerability detection. Hence they did not mention it in this context. That saying, recent research has also focused on the impediments of adopting static analysis. Johnson and colleagues [19] have identified several factors that developers dislike about static analysis tools—lot of false positives, difficult I/O, hard to understand workflow, etc. These factors may actually hinder developers from using static analysis tools; static analysis may actually be used less than desired.

The Practice of Reporting Vulnerabilities. Most reporters in our buffer overflow and SQL injection study prefer posting to lists, i.e., full disclosure (Section 9.3); in fact they often post to multiple lists. Posting to vendors of vulnerable software appears to be a rational choice, but it is not followed in general, especially when reporters become more experienced (Section 6.3.2). When these publicly disclosed vulnerabilities with detailed exploits remain unfixed (Section 6.3.3), the situation gets worse.

There is a genuine problem in the way reporters and vendors communicate. Several reporters mentioned that they went public after they found that the vendors were non-responsive (“... the developers should have received my message informing about this issue. Several weeks have passed since then, and there is still no official fix. Maybe they didn’t read my message, but if that is not the reason; I just have to say that sadly it seems the security of the users is not a priority unless the lack of it causes bad reviews in the Internet.”). They consider public disclosure, sometimes with detailed exploits, as a way to pressure the vendors. However, vendors’ silence does not mean they are uninterested. Perhaps, vendors have stopped updating this software or have brought a new version. Often, they are

overwhelmed by the number of problems reported and take a lot of time to fix [16,17]. In the meantime, vulnerability information in public domain may undermine software security. In fact, publicly disclosed vulnerabilities increase the chance of a software being targeted by attackers [4].

Recommendations. Synthesizing what we have learned, we now present concrete recommendations for reporters of vulnerabilities, secure system developers, project managers, security engineering researchers, and tool vendors.

Reporters should apply fuzzing and penetration testing approaches. Since fuzzing and penetration testing are widely considered to work well in practice, all reporters—developers as well as non-developers—should embrace these approaches.

Reporters should follow a combination of approaches to detect overflows. They should adopt manual and automated fuzzing, static analyses, even manual exploration. Austin and Williams [5] reported that a combination of multiple approaches, including manual exploration, is useful—however they studied approaches for detecting web vulnerabilities.

Project managers should use code review. Open source systems should invest on introducing systematic code review. Bosu and colleagues [8] demonstrate the usefulness of code reviews to detect security vulnerabilities. Edmundson and colleagues [13] also provided positive data. This study also hints at its usefulness (Section 6.1.6).

Researchers and tool vendors should focus on improved and reusable fuzzing tools. Reporters who developed custom fuzzing tools mentioned unanimously that their tools were simple and easy to build (Section 6.2.1). Perhaps, this engineering challenge does not appeal to researchers. But this should change.

Reporters should report vulnerabilities to vendors. Although reporters perceive that vendors do not respond quickly to the vulnerabilities reported to them, vulnerabilities reported directly to vendors (non-disclosure and coordinated disclosure) have a chance to get fixed quickly and early [4]. Vendors may find it hard to follow the many channels for full disclosure, e.g., numerous public forums, blogs, Twitter, etc., thus leaving the vulnerability

unfixed (Sections 6.3.3 and 7.3.2). Coordinated disclosure provides a middle ground between helping vendors and getting recognition, and it should be adopted more.

Vendors should respond more quickly to reported vulnerabilities. Reporters expressed a general dissatisfaction about their experience in communicating with vendors (Sections 6.3.2 and 7.3.1). Vendors should look into this issue and react. Recent studies suggest that incentives provided by vendors, such as Vulnerability Rewards Programs, have been successful [15]. This may improve reporters' perception towards reporting to vendors.

Detailed exploits should only be disclosed to vendors. Fully disclosed vulnerabilities with detailed exploits in public domain may be dangerous. The experienced reporters typically do not make this mistake. In the buffer overflow study, among the 28 reporters with over five years of experience, only 4 created detailed exploit code and released in public. On the other hand, 9 relatively inexperienced reporters did this. Because of the bad consequences, reporters should report detailed exploits only to the vendors.

Beginners should be taught about how to report vulnerabilities. Publicly disclosing exploits is dangerous, a consequence that beginner reporters fail to understand (See previous recommendation). Reporting practices should be an integral part of a beginner's education.

Chapter 11

Threats to Validity

There are several threats to validity of our study; here we discuss them following the four classic tests and discuss how they have been mitigated.

External Validity. There may be a concern about generalizability of our results. This will happen if our sample of 127 responders is not be representative of the larger population of reporters. Our group of responders have diverse experience and background (Section 5). The participants possess enough diversity to allow analytic generalization. Moreover, we collected responses from random reporters. For the buffer overflow study, we collected data over two six-month periods (Section 3.2), thereby replicating our own study. When we explored for trends in the buffer overflow study, we treated the data collected from the two rounds separately and reported a trend only if there is a similarity. However, we did not follow this in the SQL injection and the cross site scripting study. Another possible threat comes from the lack of the interaction with the responders. For example, when a responder mentions a tool, it is unclear whether he/she is aware of another. However, sample size, the distribution of experience, and the detail in the responses somewhat counter this threat.

Internal Validity. Internal validity is mainly a concern for explanatory studies. Since ours is not an explanatory study, it does not have a threat to internal validity from the interpretation aspect. One problem could have come from interpreting the responses of reporters who reported a single vulnerability (specific response) versus reporters who reported multiple vulnerabilities during the study (general response). Reporters were always asked about a specific vulnerability discovery; for multiple reporters we asked for the last vulnerability (Section 3.4). Another problem could have arisen from efficiently handling and analyzing

the results. We used TAMS Analyzer, an automated tool, which greatly reduced the data storage and interpretation problems.

Construct Validity. The success of our study is dependent on asking the right questions. We performed a pilot study to adapt the questionnaire (Section 3.4). Although the pilot study was small, it helped us significantly change the design. Most of the study questions remained the same across all rounds of data collection (Section 3.4) although we monitored the responses to see if the questions need to be rephrased; it was never necessary.

Reliability. Because we dealt with open ended questions and answers, there is always a chance of misinterpretation of data. There are three reasons why this is not an issue. First, the coding was done in two phases, once per question and once per document, to improve the reliability of codes. Second, we used structural coding [27] with codes that had clear partitions (Section 3.5); this reduces the chance of coding biases. Most importantly, coding was done by the two authors (investigator triangulation [11, 25]). We calculated Krippendorff's alpha [20] value on the initial code; it was high (e.g., 0.8339 for the buffer overflow study). Even then, the two coders analyzed the codes that differed and reached consensus.

Chapter 12

Future Work and Conclusion

Ours is the first study to collect information from a previously untapped source. It finds important details about how reporters detect three important kinds of security vulnerabilities, how reporters analyze their findings, and how they report detected vulnerabilities. In the future, we want to expand on the study and include interviews to get qualitative information about each of the phases. Section 9 attempted to explain some of the observed results. We plan to perform detailed empirical studies on each of the issues.

Our random group of participants reach consensus about the method and tools that work best. Not only that, they demonstrate emerging group behavior. A beginner searching for a guideline on how to detect vulnerabilities will now know what he/she should do.

Bibliography

- [1] O. Alhazmi and Y. Malaiya. Prediction capabilities of vulnerability discovery models. In *RAMS'06*. IEEE Computer Society, 2006.
- [2] P. Anbalagan and M. Vouk. Towards a unifying approach in understanding security problems. In *ISSRE'09*. IEEE Press, 2009.
- [3] W. Arbaugh, W. Fithen, and J. McHugh. Windows of vulnerability: A case study analysis. *Computer*, 33(12):52–59, Dec. 2000.
- [4] A. Arora, R. Krishnan, R. Telang, and Y. Yang. Impact of vulnerability disclosure and patch availability - An empirical analysis. In *WEIS '04*, 2004.
- [5] A. Austin and L. Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *ESEM '11*, 2011.
- [6] D. Baca, B. Carlsson, K. Petersen, and L. Lundberg. Improving software security with static automated code analysis in an industry setting. *Software—Practice and Experience*, 43(3):259–279, 2013.
- [7] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, Feb. 2010.
- [8] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, page To appear, 2014.
- [9] H. Browne, W. Arbaugh, J. McHugh, and W. Fithen. A trend analysis of exploitations. In *IEEE S&P '01*. IEEE Computer Society, 2001.
- [10] M. Cova, C. Leita, O. Thonnard, A. Keromytis, and M. Dacier. An analysis of rogue AV campaigns. In *Proceedings of the 13th international conference on Recent advances in intrusion detection*, RAID'10, pages 442–463, Berlin, Heidelberg, 2010. Springer-Verlag.
- [11] N. Denzin. *The Research Act: A Theoretical Introduction to Sociological Methods*. McGraw-Hill, New York, 1978.
- [12] A. Doupé, M. Cova, and G. Vigna. Why Johnny can't Pentest: An analysis of black-box web vulnerability scanners. In *DIMVA '10*. Springer, 2010.
- [13] A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner. An empirical study on the effectiveness of security code review. In *ESSoS '13*, volume 7781 of *Lecture Notes in Computer Science*, pages 197–212. Springer Berlin Heidelberg, 2013.

- [14] M. Fang and M. Hafiz. Discovering buffer overflow vulnerabilities in the wild: an empirical study. In M. Morisio, T. Dybå, and M. Torchiano, editors, *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, page 23. ACM, 2014.
- [15] M. Finifter, D. Akhawe, and D. Wagner. An empirical study of vulnerability rewards programs. In *USENIX Security '13*. USENIX Association, 2013.
- [16] S. Frei, D. Schatzmann, B. Plattner, and B. Trammell. Modelling the security ecosystem- The dynamics of (in)security. In *WEIS '09*, 2009.
- [17] S. Frei, B. Tellenbach, and B. Plattner. 0-day patch - Exposing vendors' (In)security performance. BlackHat Europe, 2008.
- [18] R. Gopalakrishna and E. Spafford. A trend analysis of vulnerabilities. Technical report, CERIAS, 2005.
- [19] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE '13*. ACM, 2013.
- [20] K. Krippendorff. *Content Analysis: An Introduction to Its Methodology*. Sage Publications Ltd, Singapore, 2004.
- [21] L. Layman, L. Williams, and R. Amant. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *ESEM '07*. IEEE Computer Society, 2007.
- [22] F. Massacci and V. Nguyen. Which is the right source for vulnerability studies?: An empirical analysis on mozilla firefox. In *MetriSec '10*. ACM, 2010.
- [23] P. Mell, K. Scarfone, and S. Romanosky. CVSS: A complete guide to the Common Vulnerability Scoring System Version 2.0. Technical report, FIRST.org, 2007.
- [24] H. Okhravi and D. Nicol. Evaluation of patch management strategies. *International Journal of Computational Intelligence: Theory and Practice*, 3:109–117, 2008.
- [25] M. Patton. *Qualitative Research & Evaluation Methods*. Sage Publications Ltd, Singapore, 3 edition, 2001.
- [26] N. Rutar, C. Almazan, and J. Foster. A comparison of bug finding tools for Java. In *ISSRE '04*. IEEE Computer Society, 2004.
- [27] J. Saldana. *The Coding Manual for Qualitative Researchers*. Sage Publications Ltd, Singapore, 2009.
- [28] B. Schneier. Full disclosure and the window of exposure. *Crypto-Gram Newsletter*, Sep 2000.
- [29] T. Scholte, D. Balzarotti, and E. Kirda. Quo vadis? A study of the evolution of input validation vulnerabilities in web applications. In *FC'11*. Springer-Verlag, 2012.
- [30] G. Schryen. A comprehensive and comparative analysis of the patching behavior of open source and closed source software vendors. In *IMF*, 2009.
- [31] SecurityFocus. Bugtraq vulnerability list. <http://www.securityfocus.com/>.

- [32] M. Shahzad, M. Shafiq, and A. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *ICSE '12*. IEEE Press, 2012.
- [33] L. Suto. Analyzing the effectiveness and coverage of Web application security scanners. Technical report, eEye Digital Security, 2007.
- [34] R. Telang and S. Wattal. An empirical analysis of the impact of software vulnerability announcements on firm stock price. *IEEE Trans. Softw. Eng.*, 33(8):544–557, Aug. 2007.
- [35] TippingPoint. Zero Day Initiative (ZDI). <http://www.zerodayinitiative.com/>.
- [36] Verisign. iDefense security intelligence services. http://www.verisigninc.com/en_US/products-and-services/network-intelligence-availability/idefense/index.xhtml.
- [37] M. Weinstein. TAMS Analyzer for Macintosh OS X: The native open source, Macintosh qualitative research tool.
- [38] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS '03*. The Internet Society, 2003.
- [39] Y. Wu, R. Gandhi, and H. Siy. Using semantic templates to study vulnerabilities recorded in large software repositories. In *SESS '10*. ACM, 2010.
- [40] R. Yin. *Case Study Research: Design and Methods*. Sage Publications Ltd, Singapore, 2004.
- [41] S. Zhang, D. Caragea, and X. Ou. An empirical study on using the national vulnerability database to predict software vulnerabilities. In *DEXA '11*. Springer, 2011.